TSOtool: A Program for Verifying Memory Systems Using the Memory Consistency Model

Sudheendra Hangal[†], Durgam Vahia[‡], Chaiyasit Manovit[‡],
Juin-Yeu Joseph Lu[‡] and Sridhar Narayanan[‡]
Processor and Network Products, Sun Microsystems
tsotool@sun.com

[†]Sun Microsystems India Private Limited
Divyashree Chambers, Shantinagar
Bangalore, 560 025 KA India

[‡]Sun Microsystems
430, N. Mary Ave
Sunnyvale, CA 94085 USA

ABSTRACT

In this paper, we describe TSOtool, a program to check the behavior of the memory subsystem in a shared memory multiprocessor. TSOtool runs pseudo-randomly generated programs with data races on a system compliant with the Total Store Order (TSO) memory consistency model; it then checks the results of the program against the formal TSO specification. Such analysis can expose subtle memory errors like data corruption, atomicity violation and illegal instruction ordering.

While verifying TSO compliance completely is an NP-complete problem, we describe a new polynomial time algorithm which is incorporated in TSOtool. In spite of being incomplete, it has been successful in detecting several bugs in the design of commercial microprocessors and systems, during both pre-silicon and post-silicon phases of validation.

Keywords

Memory consistency models, Multiprocessor verification, Sequential Consistency, Total Store Order

1. Introduction

The memory subsystem is amongst the most complex parts of modern computer system designs based on shared memory multiprocessing. Therefore, it is also among the most bug-prone. With the large gap between processor and memory speeds, and the trend towards multiple logical processors on a single chip - for example, by employing chip multiprocessing (CMP) or simultaneous multithreading (SMT) techniques - there is intense pressure on computer architects to design high performance memory systems to feed these processors. This leads to ever more complex designs involving shared caches, pipelined protocols, speculative memory operations and elaborate coherence mechanisms. Verifying that these designs work correctly, both in terms of protocol and implementation, is a challenging problem. This is not a problem for high-end workstation and server systems alone; even personal computers are beginning to incorporate multiprocessing capabilities. Our experience has been that, despite extensive simulation, bugs are still found in multiprocessing functionality after tapeout, and often take a long time to wring out.

A part of the problem in verifying multiprocessor memory systems is the difficulty of reasoning about the validity of results of a program which has data races. Since the results of such a program are timing-dependent, multiple legal outcomes may exist, and a simple architectural model of the processor cannot be used to cross-check results. TSOtool is a dynamic testing tool which is aimed at solving this problem[1]. It operates by running a pseudo-randomly generated program with data races on the system, observing the results and then checking the observed outcome for validity under the memory model of the machine (Total Store Order, or TSO). TSOtool is able to perform end-to-end checks on a detailed simulation model of the system, or on a real system, using a large space of randomly generated test cases. This approach can expose bugs in the design of the memory system no matter where they may be hiding - for example, in the design of caches, coherence protocols, system interconnects, or memory controllers.

TSOtool uses the memory consistency model of the multiprocessor system to verify the implementation of the memory system. A memory consistency model (interchangeably referred to as a memory model in this paper) is a specification of the required semantics and ordering of various memory operations on a shared memory multiprocessor. Memory models effectively establish a contract between the programmer and the machine, and therefore both programs and hardware implementations are required to be correct with respect to this definition. Memory models significantly impact the ease of programming the machine, as well as the set of hardware and compiler optimizations which may be performed legally. Adve and Gharachorloo's tutorial surveys many of the issues surrounding memory models [1]. Commercial architectures support a variety of memory models, such as Sequential Consistency (SC), Total Store Order (TSO) and Release Consistency (RC). While SC and TSO present a more intuitive model to a programmer, multiprocessors supporting these models need to perform aggressive optimizations to perform comparably to those with more relaxed models [8][10]. Making the several complex elements involved in the design of the memory

[1]In this paper, we use the term verification to refer to functional testing of hardware designs, following industry-standard terminology.

hierarchy work together to preserve the programmer guarantees afforded by the memory model is a major challenge for computer architects today.

TSOtool can be run on commercially available SPARC architecture based platforms running a standard operating system, and does not need any modifications to either the hardware or the operating system. As a result, we have been able to use it easily and effectively on a variety of multiprocessor systems based on several different SPARC microprocessors. In addition, TSOtool has been used extensively in pre-silicon validation environments. In such environments, TSOtool can optionally use any extra observability to improve the quality of results. In later sections, we will report our experiences using TSOtool, and describe the kinds of bugs we successfully found in the design of several microprocessors and multiprocessor systems, both in the microarchitecture definition and in the implementation of the microarchitecture. With minor extensions, the same approach can be used to test for compliance of test program runs to other memory models as well.

There have been several prior approaches to the verification of memory systems. With formal approaches, verifying that a particular optimization is correct under a given memory consistency model can involve subtle proof methodologies, using automatically or manually generated proofs. Such approaches usually employ a high-level abstraction of the real design to check specific properties of the abstracted implementation. However, they leave the actual implementation of the processor unchecked, which is, in fact, a significant source of complexity and errors in large designs. On the other hand, prior testing-based approaches for multiprocessors are able to test only programs whose results can be reasoned about a priori.

Our major contributions in this paper are the following:

1. We present a new polynomial time algorithm for detecting TSO violations in a multiprocessor program. With trivial modifications, this algorithm can be extended to other memory models such as SC or Partial Store Order (PSO). The algorithm is sound but incomplete from the point of view of detecting TSO violations, i.e., it will never falsely report errors, but may sometimes miss errors. We presume the machine innocent, unless proved guilty. Our algorithm is incomplete, trading off accuracy for runtime, since the problem of detecting a TSO violation completely is NP-complete.
2. We describe how we modeled a real-life instruction set (SPARC V9) in terms of the load and store operations in the formal TSO specification.
3. We describe our experiences with a new verification methodology based on TSOtool which helped us detect several subtle bugs in the design of commercial microprocessors and multiprocessor systems.

The rest of this paper is organized as follows. Section 2 describes the TSO memory model in terms of its formal axioms. Section 3 provides an overview of the different phases of TSOtool and how various instructions are modeled by TSOtool. Section 4 describes the analysis algorithm. Section 5 presents the results of using TSOtool, and describes some bugs TSOtool was able to uncover. Section 6 surveys related work, and Section 7 summarizes the paper, and concludes.

2.The TSO Memory Model

The axioms of the TSO memory model have been formally described by Sindhu et al [17]. We briefly reproduce the six axioms below. The notation used is as follows:

L_a^i	a Load to location *a* by processor *i*
S_a^i	a Store to location *a* by processor *i*
$[L_a^i ; S_a^i]$	a Swap to location *a* by processor *i*
$Val[L_a^i]$	the value read by L_a^i
$Val[S_a^i]$	the value written by S_a^i
Op_a^i	either a load or a store

Two kinds of orders are used in the definition of these axioms (an order is defined as a relation that is reflexive, anti-symmetric and transitive): a per processor program order denoted by the character ; and a global memory order denoted by the character $\leq$. In the following axioms, loads are represented in the global order by the time at which their return value is effectively bound (i.e. cannot be changed) while stores are represented by the time at which the store is effectively visible to all processors in the system. The following are the 6 TSO axioms:

Order: There is a total order over all stores.

$$\forall S_a^i, S_b^j : (S_a^i \leq S_b^j) \vee (S_b^j \leq S_a^i)$$

Atomicity: Atomicity requires that there be no intervening stores between the load and store components of an atomic operation.

$$[L_a^i ; S_a^i] \Rightarrow (L_a^i \leq S_a^i) \wedge (\forall S_b^j : S_b^j \leq L_a^i \vee S_a^i \leq S_b^j)$$

Termination: All stores and swaps eventually terminate. This is formally specified by requiring that if one processor does a store and another processor repeatedly does loads to the same location, there will eventually be a load that succeeds S in $\leq$

$$S_a^i \wedge (L_a^j ;) \infty \Rightarrow \exists L_a^j \in (L_a^j ;) \infty \; such that \; S_a^i \leq L_a^j$$

LoadOp: If an operation follows a load in ; then it must also follow the load in $\leq$

$$L_a^i ; Op_b^i \Rightarrow L_a^i \leq Op_b^i$$

StoreStore: If 2 stores appear in a particular order in ; then they must also appear in the same order in $\leq$

$$S_a^i ; S_b^i \Rightarrow S_a^i \leq S_b^i$$

Informally, the LoadOp and StoreStore axioms together imply that the only kind of reordering allowed between operations on the same processor is for loads to overtake stores, i.e. a load which succeeds a store in program order may precede it in global order.

Value: The value returned by a load is the value written to it by the last store in global order, amongst the set of stores preceding it in either global order or program order.

$$Val[L_a^i] = Val[\underset{\leq}{Max}[\{S_a^k | S_a^k \leq L_a^i\} \cup \{S_a^i | S_a^i ; L_a^i\}]]$$

The value axiom allows a load to read the value written by an earlier store on the same processor, before that store has completed in global order. Consider a load which returns the value written by an earlier store (in program order) on the same processor to the same address; this load may be ordered either before or after the store in global order. This permits implementations with store buffers to locally bypass data from a store to a load, before the store is globally visible.

In addition to these axioms, we add an axiom pertaining to memory barriers *M:*

$$Op1 ; M ; Op2 \Rightarrow Op1 \leq Op2 \; (Membar)$$

The basic TSO axioms do not refer to non-uniformly sized load and store accesses. We apply these axioms at a byte level and subject to the definition in the architecture, require that accesses of size more than a single byte happen atomically - for example, in the SPARC architecture, a single aligned memory access of up to 64 bits is required to happen atomically [20].

Real-life instruction sets have more complexity than just loads, stores and swaps. In TSOtool, we also incorporate several other memory instructions into the same framework: apart from different size loads and stores, we accommodate other kinds of atomic instructions, memory barriers, block memory operations, prefetches, non-faulting loads, non-cacheable accesses with or without side-effect, etc. An example of the kinds of ordering rules between various types of memory operations in a real processor can be found in the UltraSPARC-III Users Manual [18].

3. TSOtool Operation

There are 3 phases of TSOtool operation, as illustrated in Fig. 1.

In Step 1, TSOtool generates a pseudo-random, multithreaded test program with data races to a relatively small number of shared memory locations. Various properties of the generated program can be controlled by a user to target the test towards specific kinds of instruction sequences or sharing patterns.

In Step 2, the user runs this test program on a platform which supports the TSO memory model. If real hardware is available, this environment can be an actual multiprocessor system, running an operating system. Or, it can be a simulation model of the processor or the system. The simulation models can be at different levels of abstraction, such as architectural, RTL (Register Transfer Level) or gate-level. The simulation may model either the entire processor or only units belonging to the memory subsystem. The verification environment itself can include software simulators, hardware accelerators or FPGA-based emulation machines. We have run TSOtool generated test programs in all of these environments.

In Step 3, the results of the test program are fed back into TSOtool for analysis. At the end of analysis, a pass or fail is signaled. Note that it is possible that different runs of the same test program may observe different results in the presence of external perturbation (such as operating system activity.) Therefore, the analysis result always applies to the correctness of a particular run of the test program.

The rest of this section describes each of these phases in more detail.

3.1 Test Generation

In the test generation phase, TSOtool creates a pseudo-random program with data races, based on optional inputs from a user. Users typically get the generator to create a relatively short test with intense sharing. Users can control parameters such as the relative frequency of instruction types, memory layout and loop characteristics. Based on these parameters, TSOtool generates an internal representation of the test program, each thread represented by a sequence of nodes corresponding to every operation in that thread. This program sequence is then mapped to either a set of assembler instructions, or a series of instructions in some other language suitable for the test environment. Occasionally, we need to randomize events during the test (such as the direction of hard-to-predict conditional branches), so a dynamic software LFSR is maintained on each processor and used as a source of random numbers.

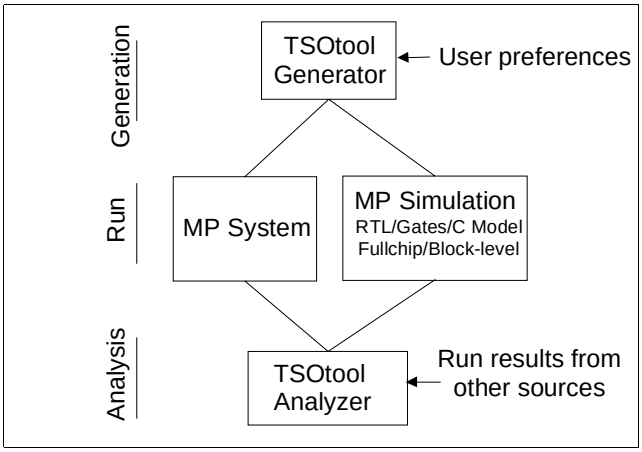

Fig.1: TSOtool usage flow

Unique store values: Having unique store values in the test program helps TSOtool map every load value back to the store which created it. This feature is important for the analysis algorithm, as explained in Section 4. We ensure that store values are unique by maintaining two running counters, one each in a floating point register and an integer register. These counters are used as the source of store values in the test program. The expense of maintaining these counters is minimal - an increment operation for every unique store value.

Load Observability: On physical systems, which provide no additional observability, the test program includes code to observe and save the results of all the load operations in the program. The results are initially buffered in two sets of processor registers, one for floating point results and one for integer results. When a results buffer is full, its contents are flushed to memory. Buffering helps to reduce perturbation in the middle of test operations. In environments where the load results can be observed through other means, code to explicitly save results may not be needed.

Other instructions: In addition to 32-bit, 64-bit and 128-bit loads and stores, some of the other kinds of operations supported by the generator are the following:

- Memory access instructions to various Address Space Identifiers (ASIs)
- Memory barrier instructions - these require that all previous instructions on the issuing processor are globally visible before the next instruction is issued.
- Various flavors of prefetch, such as prefetch for read-once, write-once, read-many, or write-many. Prefetches may be 'strong' or 'weak'. Strong prefetches may incur TLB miss traps, while weak prefetches are silently dropped in case of a TLB miss. Certain patterns of load accesses can also trigger a hardware prefetch operation in some processors.
- Different types of block load and store instructions which read or write 64 bytes at a time. These have special rules to ensure ordering with respect to other instructions.

- Instructions which flush data from various levels of the cache, or instructions which flush the execution pipeline.
- Compare and swap instructions: Compare and swap (CAS) instructions are emitted with a preceding load of the same size to the same address. The value returned by the load is used as the compare value for the CAS instruction. This gives the CAS a reasonable probability of resolving into a swap; the compare may occasionally fail when a store to the same address intervenes between the load and the CAS instructions.
- Non-faulting loads: These are loads which silently return 0 if the address causes a memory fault. For valid memory addresses, the behavior is required to be the same as that of a regular load. Non-faulting loads in the test program are randomly marked to access either faulting or non-faulting addresses.
- Unpredictable conditional branches
- Sequences of operations which cause cache line replacements and writebacks.
- Inter-processor interrupts

TSOtool allows users fine-grained control over the test program, as well as the ability to specify desirable sequences of memory operations which are considered likely to exercise known corner-cases in the design, such as a queue in the system becoming full or a hazard condition being created. Users can improve the quality of testcases generated using tools which report test coverage.

3.2 Test Run

As mentioned earlier, the generated test program can be mapped to a variety of test environments. On physical systems, we typically run TSOtool on configurations of up to 16 processors with a few thousand memory operations per processor.

In a simulation environment, TSOtool can optionally utilize the additional observability provided by the environment. For example, if the result of load operations can be directly observed from the simulation, explicit operations to buffer and save them are omitted from the test program.

Simulation environments often have the useful capability to detect errors via runtime checkers monitoring the design. TSOtool can make use of these checkers to detect failures in the course of simulation. In some accelerated simulation environments, however, it is expensive or impossible to observe events in the system or to add runtime checkers. In one such environment, we can improve simulation throughput by a few orders of magnitude by disabling observability features and runtime checkers. In these cases, TSOtool's ability to independently observe the results and analyze them for correctness is very useful.

3.3 Analysis

In the analysis phase, the nodes in the program representation of step 1 are first expanded to form nodes in an analysis graph. The analysis graph is formed by unrolling loops and resolving branches in the original program to model the dynamic sequence of memory operations in the test. Nodes representing instructions which cover multiple shared words of interest are expanded, so that all loads, stores and swaps in the analysis graph are of a uniform size.

Before starting analysis, the remaining program nodes are processed in the following manner:

- Prefetch instructions, cache or pipeline flushes and cache line replacements and writebacks should have no programmer visible effect and are ignored for the purpose of analysis.
- Non-faulting loads to illegal addresses are checked for a return value of 0, and then ignored for the rest of the analysis. Non-faulting loads to legal addresses are converted to regular loads.
- Compare and swap instructions are resolved by examining the return value of the instruction. If the CAS completed, the instruction is converted to a swap of the same size, else it is converted to a regular load.

Next, edges are added in this graph to represent the memory order $\leq$ according to the algorithm described in the next section.

The TSOtool analyzer also has a standalone analysis interface through which it can be fed a program description along with the values of all loads and stores, and this outcome can be checked for TSO violations. This feature allows us to potentially plug in the results from other test programs which obey the unique store values requirement.

3.4 Debug

When a TSO violation is detected, TSOtool emits a graphical representation of the relevant area in the analysis graph. The user can click on each edge in the graph to understand the reason for its existence, and hence follow the chain of reasoning used by TSOtool to infer the edge.

TSOtool also emits the analysis graph to a text file in a format comprehensible to users. Users can edit this file and feed it back to TSOtool via the analysis interface if they wish to make an educated guess about which load result is incorrect and what the correct load result should have been. This “what-if” analysis is often useful to evaluate the correctness of other possible results.

4. Analysis Algorithm

The TSOtool analyzer is the key component that differentiates our approach from conventional approaches used in multiprocessor verification. The analyzer reads in the sequences of memory operations on all processors, annotated with the result returned by each load during program execution. It then infers as many relations as possible between memory operations that must hold in order to satisfy the TSO axioms.

A directed graph is used as the data structure for the analysis. Nodes in this graph represent operations and edges represent ordering relations in the global memory order $\leq$. Since $\leq$ is transitive, any path in the graph implies the existence of the $\leq$ relation between the source and destination of the path. We ignore reflexivity of $\leq$ by not adding an edge from each node to itself.

A synthetic node at the root of the graph acts like a set of stores writing initial values to all shared addresses. A violation of any TSO axiom will cause a conflict in the order of two or more operations and manifest as a cycle in the graph. A cycle implies that $\leq$ is not a valid order.

```
Input:
A per processor instruction sequence consisting of loads, stores, and
membars. A swap is considered to be both a load and a store.
A function map, which maps a value to the store which created it:

Output:
A boolean value indicating whether or not the given program outcome
obeys all the TSO axioms.

[rule R1-R3]
for each processor
  for each instruction node n in the program order
    if n is a store or a membar then
      add edges from last load, store, and membar to n
    else if n is a load then
      add edges from last load and membar to n
    end if
  end for
end for

[rule R4]
for each load instruction L
  S := map(load value of L)
  if S does not precede L in program order then add edge S→L
end for

[rule R5]
for each load L,
  S' := last store to this address preceding this L in program order
  S := map(load value of L)
  if S != S' then add edge S'→S
end for

[rule R6 and R7] - done in iterations
do
  for each load L
    S := map(load value of L)
    recursively trace all store predecessors S' of L:
      if S' != S and they write to the same address then
        add edge S'→S
      end if
      flag a TSO violation if a cycle is found
  end for
  for each store S
    recursively trace all store successors S' of S:
      if S' and S write to the same address then
        add edge L→S' for all loads L reading value written by S
      end if
      flag a TSO violation if a cycle is found
  end for
until no more edges can be added

find cycles in the graph and flag a TSO violation if a cycle is found
```

Fig. 2: High level description of TSOtool's analysis algorithm

A set of atomic operations is modeled in the graph by forcing incoming edges incident to any node in the set to point to its first node; outgoing edges from any node in the set similarly leave from its last node.

Fig. 2 provides a high level outline of the TSOtool analysis algorithm. The algorithm uses as input a mapping of every value to the store which wrote that value. A load reading a value never written to that address is signaled as a failure at the outset. After this step, the algorithm adds edges by applying the following rules.

Static Edges: In the first step, program order edges are added to the graph according to the following 3 rules. These edges are independent of run results:

1. R1: $L;Op \Rightarrow L \leq Op$ (LoadOp axiom)
2. R2: $S;S' \Rightarrow S \leq S'$ (StoreStore axiom)
3. R3: $S;M;L \Rightarrow S \leq L$ (Membar axiom)

The above rules ensure that the LoadOp, StoreStore and Membar axioms are satisfied by the relations embodied in the graph.

For the remaining rules, let S, S', and L be accesses to the same address.

Observed Edges: For all loads, the edges specified by the following two rules are added based on the load results.

4. R4: $Val[L]=Val[S] \wedge \neg\, S;L \Rightarrow S \leq L$ (Value axiom)
   This follows because S must be in one of the two store sets in the Value axiom for L
5. R5: $Val[L]=Val[S] \wedge S';L \Rightarrow S' \leq S$ (Value axiom)
   This must be true because if both $S \leq S'$ and $S';L$ are true, $Val[L]$ cannot equal $Val[S]$ by the Value axiom.

Inferred edges: In the last step, we add more edges based on two rules which follow from the Value axiom:

6. R6: $Val[L]=Val[S] \wedge S' \leq L \Rightarrow S' \leq S$ (Value axiom)
   Assuming otherwise, $S \leq S'$ (and given $S' \leq L$) will lead to a contradiction because $Val[L]$ cannot equal $Val[S]$.
7. R7: $Val[L]=Val[S] \wedge S \leq S' \Rightarrow L \leq S'$ (Value axiom)
   Assuming otherwise, $S' \leq L$ (and given $S \leq S'$) will lead to a contradiction because $Val[L]$ cannot equal $Val[S]$.

For rule R6, the set of all possible S' such that $S' \leq L$ can be found by traversing the graph backward from L to find its predecessors known at that time. Similarly for rule R7, traversing the graph forward from S will reach all S' such that $S \leq S'$. However, this forward and backward graph traversal depends on predecessors and successors of the nodes in global order, which is still in the process of being derived. To overcome this problem, we iterate over the application of rules 6 and 7 to the graph, till a fixed point is reached and no further edges are added in a complete iteration. The graph is then checked for cycles. If a cycle exists, it implies that the relations derived do not constitute a valid order.

Fig. 3 illustrates an example of a 4-thread program outcome which violates TSO. There are 2 memory locations involved: A and B. The notation for this example and the following examples in this paper is: S[A]#1 refers to a store which writes value 1 to location A, while L[B]=92 refers to a load to address B which reads value 92. Fig. 4 illustrates graphically how a cycle is formed in the analysis graph.

P1	P2	P3	P4
S[B]#91	*S[A]#2*	*S[B]#92*	*L[B]=92*
S[A]#1		*L[A]=2*	*L[B]=91*
L[A]=2		*L[B]=92*	

Fig: 3: An Example Program outcome which violates TSO

– First, static edges E1, E2, and E3 are added using rules R1 and R2 which simply establish program order relationships using the LoadOp and StoreStore axioms.

– Next, observed edges E4 to E7 are added by applying rule R4 to all load nodes in the graph. Note that rule R4 does not create an edge from S[B]#92 to L[B]=92 on P2 because they are on the same processor.

– Next, observed edge E8 is added by applying rule R5 to L[A]=2 on P0

– Next, during application of Rule 6 for the load L[B]=92 on P2, S[B]#91 is found to be one of its predecessors (this is due to the presence of edge E8); this lets us add inferred edge E9.

– Finally, tracing the predecessors of L[B]=91 on P3 leads us to S[B]#92, giving us the inferred edge E10 (rule R6).

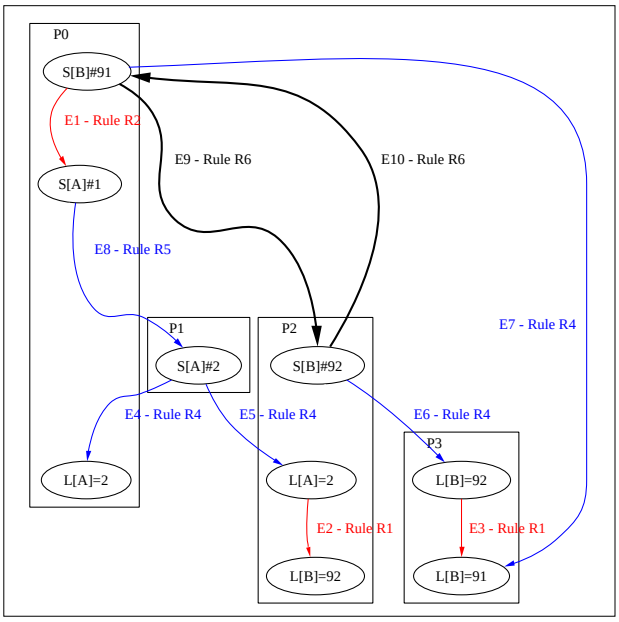

Fig. 4: Edges inferred by TSOtool for the program in Fig. 3

A cycle in the graph (shown in bold) is formed by edges E9 and E10 indicating a conflicting order between S[B]#91 and S[B]#92, a TSO violation.

Time Complexity: Verifying Sequential Consistency (VSC) is known to be an NP-Complete problem even when the mapping function between load values to stores which wrote those values is known. This is termed as the VSC-read problem by Gibbons and Korach [9] – in this terminology, our problem would be called the VTSO-read problem (Verifying TSO with read-mapping). Since every instance of a VSC-read problem can be trivially mapped to an instance of the VTSO-read problem by inserting memory barriers after every store which is succeeded by a load in program order, it can be shown that the VTSO-read problem is also NP-complete.

The algorithm in Fig. 2 clearly runs in polynomial time: if the number of nodes in the graph is n, the number of iterations is bounded by the number of all possible edges, $O(n^2)$ since each iteration adds at least one edge. The time complexity of each iteration is at most $O(n^3)$ since there are $O(n)$ Store-Load pairs, and we need to spend at most $O(n^2)$ time to traverse each edge in the whole graph for each pair. In practice, we implement optimizations to bound the predecessor and successor subgraph traversal when it is known that no new constraints can be added to the graph. Our analysis algorithm runs in the order of minutes on programs with about 100,000 operations – Section 5 has more details on analysis runtime.

The analysis algorithm described in this section can also be easily modified to verify other memory models. For example, in Sequential Consistency, all the rules remain the same, except for the additional requirement between stores and loads on the same processor.[2] Therefore, the only difference lies in the initial set of edges determined from program order and the application of the remaining rules remains the same.

Incompleteness: In the absence of cycles in the graph, our polynomial time algorithm creates a global order relation which is consistent with the LoadOp, StoreStore, Membar, Value and Atomicity axioms. (The Termination axiom involves an infinite sequence of loads and is difficult to test in practice; but it is vacuously satisfied.) However, our algorithm is incomplete because it does not explicitly ensure that the Order axiom is satisfied. To satisfy the Order axiom, we would have to identify unordered writes at the end of our algorithm, and search for a combination of relations between them which is compatible with the results; this search would make the runtime exponential in the worst case, and the analysis time impractically large. By not explicitly enforcing the Order axiom, our algorithm trades off accuracy for reasonable analysis time.

Fig. 5 illustrates a case where an existing relation is not inferred by our analysis algorithm; the edges in the graph are depicted at the point that the algorithm of Fig. 2 has reached a fixed point and terminated. Notice that S[A]#1 and S[A]#2 are left unordered. However, we can reason that S[A]#1 ≤ S[A]#2 must be true. If not, S[A]#2 ≤ S[A]#1 by the Order axiom; but given this order, only one of the two values, either 3 or 4, can be read by the two loads to location B that are ordered after S[A]#2. While this example is not yet a missed TSO violation, adding a similar, mirrored set of nodes to a different location C (2 stores to C ordered before S[A]#1, and 2 loads to C ordered after S[A]#2) creates an instance of a TSO violation which is missed by our algorithm.

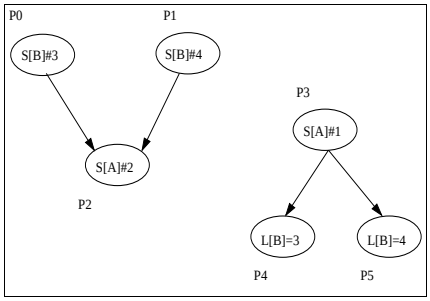

Fig. 5: An example when TSOtool misses an edge

[2]The Value axiom for the SC memory model is defined as: $Val[L_a^i] = Val[\underset{\leq}{Max}\{S_a^k | S_a^k \leq L_a^i\}]$;

the exclusion of the term $\{S|\ S;L\}$ compared to the TSO Value axiom does not matter because $S;L$ implies $S \leq L$ in SC

5. Results

TSOtool has found numerous bugs during both the design simulation and silicon bringup processes on various types of processors and systems at Sun Microsystems. In this section, we present the results of deploying TSOtool on 6 different microprocessors, without explicitly identifying each one of them for proprietary reasons.

Sun currently has several teams working towards the development of both new generation and derivative microprocessors. Despite employing widely different architectural techniques, all of these processors and systems support the TSO memory model. This is not surprising since switching to a more relaxed memory model introduces potential incompatibility problems with pre-existing software, and is therefore almost as difficult as making changes to the instruction set.

A major advantage of our approach, which reasons about correctness at the memory model interface without incorporating knowledge of implementation details, is that it allows us to deploy TSOtool on several different processors and systems efficiently. Even in cases where we use the observability in simulation environments to extract load results, the dependence on the details of the environment is usually quite small. Most environments usually support a mechanism to trace the dynamic instruction sequence executed, along with the architectural results of each instruction. This is sufficient for TSOtool to obtain the load values read by the test program.

Table 1 lists the number of bugs uncovered by TSOtool on six processor designs based on the SPARC architecture. This total of 106 bugs is classified based on whether they were architecture bugs (the design worked as intended, but the microarchitecture specification itself was wrong), design bugs (the designer missed a corner case which violated the specification), and monitor or environment bugs, which exposed problems in runtime checkers or other parts of the simulation environment. Most of these bugs involved complex interaction between multiple functional units and require a detailed understanding of the design to root-cause.

CPU	**Architecture Bugs**	**Design Bugs**	**Monitor** Bugs	**Environment Bugs**
CPU1	0	3	0	0
CPU2	0	4	3	0
CPU3	0	11	8	5
CPU4	0	17	8	0
CPU5	2	20	5	0
CPU6	5	14	1	0
Total	7	69	25	5

Table 1: Classification of bugs found by TSOtool on various processors

CPU1 to CPU4 are derivative processors based on an earlier design that include significant changes and enhancements in cache hierarchy, memory controller and bus interface. The core pipeline remained unchanged in all these derivatives. In these derivatives, TSOtool did not expose architecture bugs (since the architecture was already stable), but did find bugs in the design and the verification environment. CPU5 and CPU6 are completely new designs and in these cases, TSOtool uncovered architecture bugs which were overlooked by the architects.

Table 2 shows the classification of bugs in terms of functional units. For CPUs 1-4 the presence of bugs was mainly in the cache units, memory controller and bus interface units for CPU1-4, consistent with the derivative nature of these processors.

CPU	**Pipe**	**Caches**	**TLB**	**LSU**	**Mem Cntlr**	**Interco nnect**
CPU1	0	3	0	0	0	0
CPU2	1	5	0	0	1	0
CPU3	0	17	0	0	0	2
CPU4	0	8	0	0	8	9
CPU5	3	11	6	4	0	1
CPU6	0	5	0	10	0	0
Total	4	49	6	14	9	12

LSU = Load/Store Unit

Table 2: Bugs found by TSOtool in various functional areas

5.1 Bug Examples

To illustrate the nature of bugs which TSOtool found, consider the example in Fig. 6. This illustrates relevant operations from a bug found by TSOtool during the silicon bringup process in one of the processors. BST refers to a 64-byte block store operation. A is a 4 byte memory location.

P0	P1
BST [A]#1	*SWAP [A]=1, #2* *LD [A]=1*

Fig. 6: Example of a bug found by TSOtool

(SWAP[A]=1,#2 refers to a swap to location A, reading the value 1 and writing the value 2)

The above outcome is a violation of the TSO memory model. TSOtool detects the violation for this outcome based on the following reasoning:

Application of rule R1 and the fact that SWAP is atomic add the following edge:

- SWAP $\leq$ LD

Application of rule R4 adds the following edges:

- BST $\leq$ SWAP
- BST $\leq$ LD

Application of rule R5 on the BST-LD pair adds the following edge:

- SWAP $\leq$ BST

These relations when combined create a cycle in the analysis graph, due to the contradiction in the order between the BST and the SWAP.

After careful debugging, the designers realized that the following sequence of events had lined up to create this problem. This particular processor had a write cache which acted as a buffer for writes.

1. The BST instruction on P0 caused an invalidate to be issued on the system bus for the cache line containing address A.
2. The SWAP on P1 was issued and found the corresponding line present in its write cache in the modified state, while the invalidate was still in flight.
3. The BST invalidate reached the L2 cache and write cache on P1. The swap had to re-read the updated data from main memory, and the line was installed in L2 cache with the BST data.
4. The store part of the swap wrote the new data in the write cache, but because of a design error, did not modify the tag of that line in the write cache to dirty. This led to the data update being lost when the line was later replaced in the write cache.

The lost tag write in step 4 happened due to a special optimization the processor performed when the tag is in a certain state at the time the swap is issued. Notice that this type of problem may remain latent for a long time in a system design, leading only occasionally to mysterious data corruption or system crashes. Only proactive testing using test programs with aggressive data races can expose such problems during the verification or debug phase.

Fig. 7 illustrates another real bug found on a different processor in the pre-silicon verification environment. CAS refers to the atomic compare and swap instruction. This following outcome violates the atomicity of the CAS instruction:

P0	P1
(Initial value in locations A and B is 0)	
CAS [A]=0, #1	*CAS [B]=0, #1*
LD [B]=0	*LD [A]=0*

Fig. 7: Example of a bug found by TSOtool

Both CAS instructions completed the swap successfully. Let CAS.LD refer to the load part of this instruction and CAS.ST refer to the store part. The above outcome is flagged as a TSO violation by TSOtool, based on the following inferences:

Application of rules R1 and R2 add the following program order edges:

- CAS.LD [A] ≤ CAS.ST [A]
- CAS.LD [A] ≤ LD [B]
- CAS.LD [B] ≤ CAS.ST [B]
- CAS.LD [B] ≤ LD [A]

The following edges are added due to Rule R7 and the fact that the Load and Store parts of the CAS are atomic.

- LD [B] ≤ CAS.ST [B] & LD[B] ≤ CAS.LD [B]
- LD [A] ≤ CAS.ST [A] & LD[A] ≤ CAS.LD [A]

These edges lead to a cycle in the graph. The cause is an atomicity violation in the CAS instruction.

Again, after careful debugging of this problem, the designers realized that the problem was caused by a performance optimization in the microarchitecture which had been thought to be valid. The optimization caused the lock for the atomic swap to be released early, before the store part of the swap was complete. In some cases, this optimization was incorrect and opened a window for another store to sneak in and cause an atomicity violation.

Here are root-causes of some other failures detected by TSOtool:

- A prefetch cache dropped an invalidate request, and later returned stale data to the pipeline.
- Cacheable and non-cacheable stores went through different write queues; in some cases, the ordering between these queues was violated.
- The DRAM controller dropped a speculative load request due to a buffer full condition, leading to data corruption later.
- Bus controller flow control logic caused a deadlock scenario in the system.

Besides such hardware bugs, TSOtool also found two bugs in the operating system. These bugs were related to the way the operating system emulated some of the memory instructions of the architecture. This demonstrates the immense power of an end to end checker for the complete memory system.

5.2 Analysis Runtime

Fig. 8 and Fig. 9 plot analysis runtimes for various configurations on a 450 MHz UltraSPARC-II based system. Although the actual runtime is affected most by the number of nodes in the graph, there are two other factors that affect the structure of the graph and the number of iterations required to reach a fixed point, and hence the runtime. These two factors are the processor count and number of shared locations. Fig. 8 illustrates the effect of number of processors on runtime with number of shared words fixed to 16. The X-axis in this graph represents the total number of memory operations in the test and the Y-axis represents the runtime for TSOtool analysis. A couple of points to notice in the graph are:

- Run time scales quite linearly with total memory operations for a given number of processors
- For the same number of total memory operations, run time increases with the number of processors

This is mainly because a higher number of processors creates more ordering relationships between different processors and creates a broader and denser analysis graph, which in turn takes longer time to reach a fixed point.

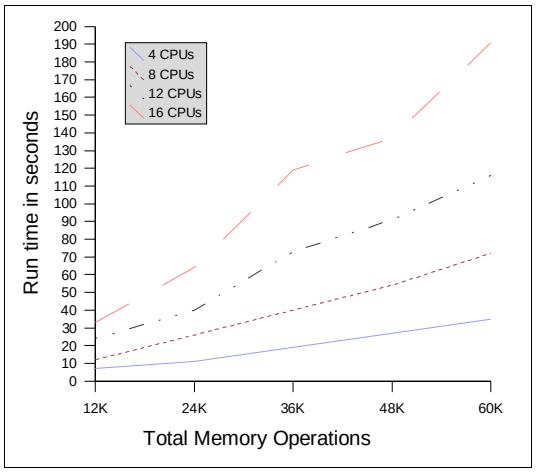

Fig. 8: Analysis time for different procesor counts

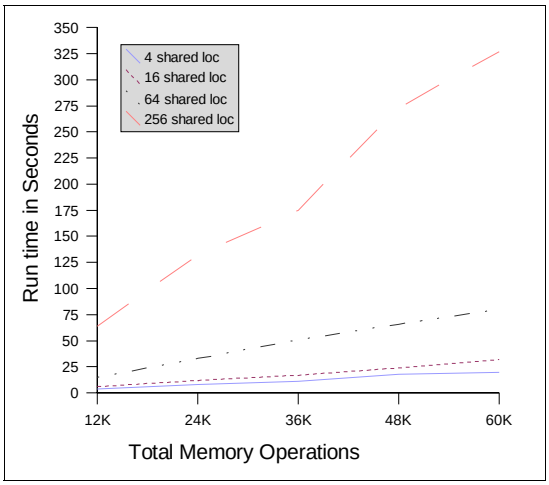

Fig. 9: Analysis time for different # shared addresses

Fig. 9 shows the effect of the number of shared locations on runtime for a fixed processor count of 4. Again, notice that runtime varies linearly with the number of memory operations for a given number of shared locations, and for a constant number of memory operations, runtime increases with a higher number of shared locations. This can be attributed to the fact that more addresses lead to a sparser graph with more dispersed ordering relations between processors. This causes a larger number of nodes to be visited during the traversal of predecessor/successor subgraphs due to Rules R6 and R7.

Since TSOtool is often able to trigger and detect problems in system-level environments using relatively short test programs, a TSOtool test failure on hardware has a good probability of being reproduced in the simulation environment. This is critical for porting the test to simulation environments, where debugging is easier but speeds are much lower than on physical hardware.

6. Related Work

Gibbons and Korach established theoretical bounds on the complexity of verifying sequential consistency under various conditions [9]. Cantin et al established similar results on the complexity of verifying memory coherence, where only one memory location is involved [6].

Other work aimed at verifying memory models in practice can be broadly categorized into static approaches and dynamic approaches. Static methods of verifying memory consistency models usually depend on formally proving that some model of the system obeys rules of the memory model [3][4]. While such methods can find bugs in protocols and optimizations at a high-level, they may miss several bugs which are present only in the implementation. The implementation is a ripe source of bugs, since a high-end microprocessor design consists of millions of lines of code. Formal approaches do not scale to large systems with a lot of detail embedded in them.

Dynamic testing on the other hand can exercise the system in all its detail, but is limited to bugs which can be uncovered by the test cases run on the system. ARCHTEST is a program which tries to identify the memory model of a multiprocessor by running a specific set of test cases which look for evidence of various kinds of ordering relaxations [7]. Nalumusu et al use the tests in the ARCHTEST framework in conjunction with model checking on a Verilog representation of part of the memory system [14]. Both of these approaches require the tests to be fixed idioms, whose outcome can be reasoned about a priori, and cannot work with pseudo-random tests.

An exhaustive approach due to Park and Dill [15] uses an executable specification to enumerate all possible outcomes for small assembler programs under a specified memory model. This approach can be applied to verify the correctness of the assembly language programs including synchronization routines; however, it does not attempt to detect faults in the hardware implementation of the memory model, and does not scale to large programs with thousands of instructions.

The idea of using a constraint graph to model relations between memory operations has been used before in the context of verifying ordering, or analyzing the performance of multi-threaded programs [5][11][16].

Industrial design teams pay a great deal of attention to memory system verification. They use random code generators extensively for processor verification [2][12][13][19]. Most code generators use a self-checking mechanism or an instruction-level simulator to check for correct execution of such programs. However, such checking usually does not work in the presence of data races in multithreaded programs. Therefore, pseudo-random code generators often have to either omit data races entirely, or control the placement of such races carefully. The only error they can check in the presence of data races is an obvious manifestation of a problem like a processor hang, or an error caught by a checker in the simulation environment. Verification approaches which try to use extra design observability present in simulations to reason about ordering and the outcome of data races are usually tied intimately to design details; they are complex to write, and often start with the assumption that the microarchitecture is correct. They are not easily portable across different processor microarchitectures, and cannot be used on physical systems where such observability is not available. In contrast, TSOtool reasons about correctness at the architectural level, and scales easily across multiple microarchitectures and multiprocessor environments both before and after silicon is available.

7. Conclusions and Future Work

We have described TSOtool, a program which verifies a machine's implementation of its supported memory consistency model. This program is very effective in detecting bugs in the design of complex microprocessors and multiprocessor systems. Many of the bugs which TSOtool found would have otherwise been extremely hard to find and debug, caused subtle crashes, undermined the reliability of the system, and been expensive to fix later.

To the best of our knowledge this is the first effort to systematically verify the entire memory system of real multiprocessor systems using the formalism of the memory model. In future, we expect to reduce the cost of TSOtool analysis further and make TSOtool failures easier to debug.

Acknowledgments

Shrenik Mehta and Mike Splain provided help in the initial stages of TSOtool development. Aleksandr Gert and Rohit Kumar have contributed substantially to this work. Hemant Gupta and Nitin Gupta performed several experiments with TSOtool. We thank several users of TSOtool who provided useful feedback.

References

[1] S. V. Adve and K. Gharachorloo, *Shared Memory Consistency Models: A Tutorial*, Digital Western Research Laboratory Technical Report, 1995

[2] B. Bentley, R. Gray, *Validating The Intel Pentium-4 Processor*, Intel Technology Journal, 1st Quarter 2001

[3] A.E. Condon and A.J. Hu, *Automatable verification of sequential consistency*, In 13th Symposium on Parallel Algorithms and Architectures, pages 113-121, ACM, 2001.

[4] A.E. Condon, M.D. Hill, M. Plakal, and D.J. Sorin, *Using Lamport Clocks to Reason About Relaxed Memory Models*, In Proceedings of the Fifth IEEE Symp. High-Performance Computer Architecture, pp. 270-278, Jan. 1999.

[5] H.W. Cain, M.H. Lipasti and R. Nair, *Constraing Graph Analysis of Multithreaded Programs*, In Proceedings of the 12the International Conference on Parallel Architectures and Compilation Techniques (PACT'03), Sept.-Oct. 2003.

[6] J.E. Cantin, M.H. Lipasti and J.E. Smith, *The complexity of Verifying Memory Coherence*, in Proceedings of the fifteenth annual ACM symposium on Parallel Algorithms and Architectures, pp. 254-255, ACM, 2003.

[7] W.W. Collier, *Reasoning About Parallel Architectures*, Prentice Hall, 1992.

[8] C. Gniady, B. Falsafi, and T. N. Vijaykumar, *Is SC+ILP =RC?*, In Proceedings of the 26th Annual International Symposium on Computer Architecture, 1999

[9] P. B. Gibbons and E. Korach, *Testing Shared Memories*, In Siam Journal on Computing, pages 1208-1244, August 1997.

[10] M.D. Hill. *Multiprocessors should support simple memory-consistency models*, IEEE Computer, pages 28-34, August 1998.

[11] A. Landin, E. Hagersten, and S. Haridi, *Race-free Interconnection Networks and Multiprocessor Consistency*, In Proceedings of the 18th Annual International Symposium on Computer Architecture, 1991.

[12] J.M. Ludden et al, *Functional verification of the POWER4 microprocessor and POWER4 multiprocessor systems*,In IBM Journal of Research and Development, 2002.

[13] S. Mehta et al, *Verification of the UltraSPARC Microprocessor*, IEEE Compcon 95, March 1995.

[14] R. Nalumasu, R. Ghughal, A. Mokkedem, and G. Gopalakrishnan, *The `Test Model-checking' Approach to the Verification of Formal Memory Models of Multiprocessors*, In Proceedings of Computer Aided 29 Verification, 10th International Conference, pages 464--476, June 1998.

[15] S. Park and D.L. Dill., *An executable specification and verifier for relaxed memory order*, IEEE Transactions on Computers, 48(2), 1999.

[16] S. Qadeer, *On the verification of memory models of shared-memory multiprocessors*, In Proceedings of the 12th International Conference on Computer Aided Verification, 2000.

[17] P.S. Sindhu, J.M. Frailong and M. Cekleov, *Formal Specification of Memory Models*, Xerox PARC Technical Report, December 1991.

[18] Sun Microsystems, *UltraSPARC-III User's Manual*, http://www.sun.com/processors/UltraSPARC-III, 2001.

[19] S. Taylor et al, *Functional verification of a multiple-issue, out-of-order superscalar Alpha processor: the Alpha 21264 Microprocessor*, In Proceedings of the Design Automation Conference, pages 638-643, 1998.

[20] D.L.Weaver, T. Germond, Editors, *The SPARC Architecture Version 9*, Prentice Hall, 1994.